

EasyBuild Module Manager

Jonathan Ghaly, Andrew Naseif*

January 2024

Contents

1	Introduction	2
2	Software Used	2
3	Environment	2
4	Database	3
4.1	ERD[1]	3
5	EasyBuild Hook[3]	4
5.1	Start Hook	4
5.2	Pre-Fetch Hook	4
5.3	Post-Fetch Hook	5
5.4	Post-Install Hook	5
5.5	End Hook	5
6	TUI (Terminal User Interface)	5
7	Logging	9
7.1	Debug Level	10
7.2	Info Level	10
7.3	Error Level	10
8	Diagrams	11
8.1	Architecture	11
8.2	Module Dependency Tree Diagram	12
8.3	Class Diagram	13
9	Future Works	14
9.1	Implementing Multi-Tenant Architecture with User Stack[4]	14
9.1.1	Expected Benefits:	14
9.2	Unused Modules	15

*Applied Innovation Center MCIT

1 Introduction

“There is no automatic way to uninstall a module. The reason is that if you remove some modules, there is (currently) no way to find out if other modules depend upon it. “[2] The unsafe way to remove a module may be to locate the module file in your `$MODULEPATH`. Examine the module’s root directory and remove the files belonging to the module. Finally, remove the module file itself. The best approach to renewing your module list is to generate a complete module tree from scratch. In this way, you can select a new and smaller set of modules to build.”

We present a comprehensive solution comprising multiple procedural steps. Initially, the process involves gathering information about the installed module. Subsequently, upon the successful installation of the module, it is seamlessly integrated into the module database along with its associated data and dependency graph.

In addition, we have developed a Terminal user interface to facilitate the management of installed modules. This interface allows users to effortlessly list installed modules along with their dependencies, providing a comprehensive overview of the system’s configuration. Furthermore, users can utilize this interface to conveniently delete modules as needed, streamlining the process of module management and ensuring system efficiency.

2 Software Used

1. MySQL
2. Python
3. EasyBuild Framework
4. Curses

3 Environment

Before proceeding, it is necessary to define certain environment variables. These variables ensure the smooth operation of the application. Here are the required environment variables:

```

1  MYSQL_PASSWORD=""                # Your MySQL password
2  MYSQL_DATABASE=easybuild         # Your MySQL database name
3  MYSQL_USER=root                  # Your MySQL username
4  MYSQL_HOST=127.0.0.1             # Your MySQL host
5  MYSQL_PORT=3306                  # Your MySQL port
6  MODULES_PATH_EB="./Modules"     # Path to the modules directory
7  LOG_PATH="/var/log/easybuildinterface.log" # Log file path

```

4 Database

4.1 ERD[1]

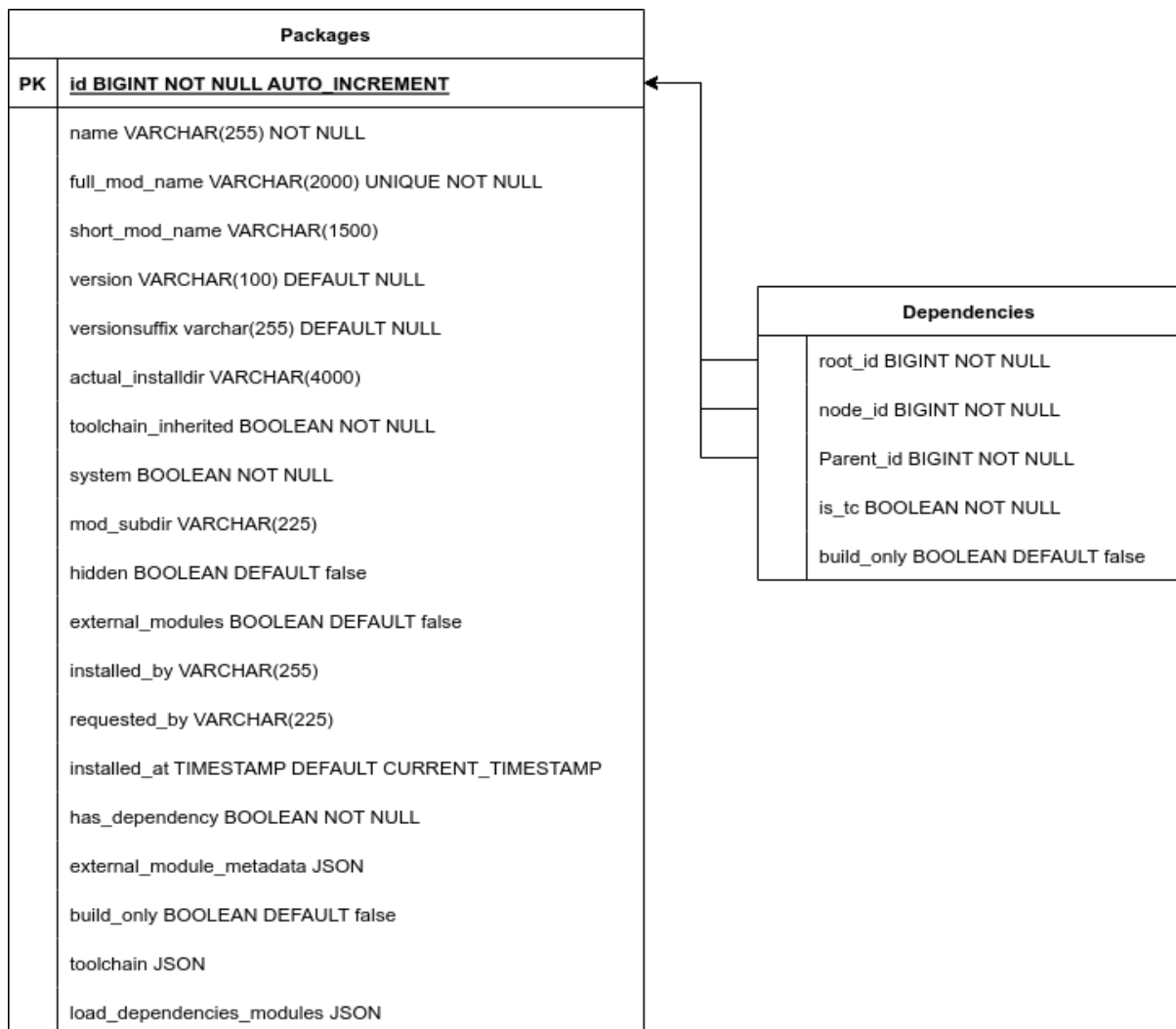


Figure 1: EasyBuild Module Manager ERD

5 EasyBuild Hook[3]

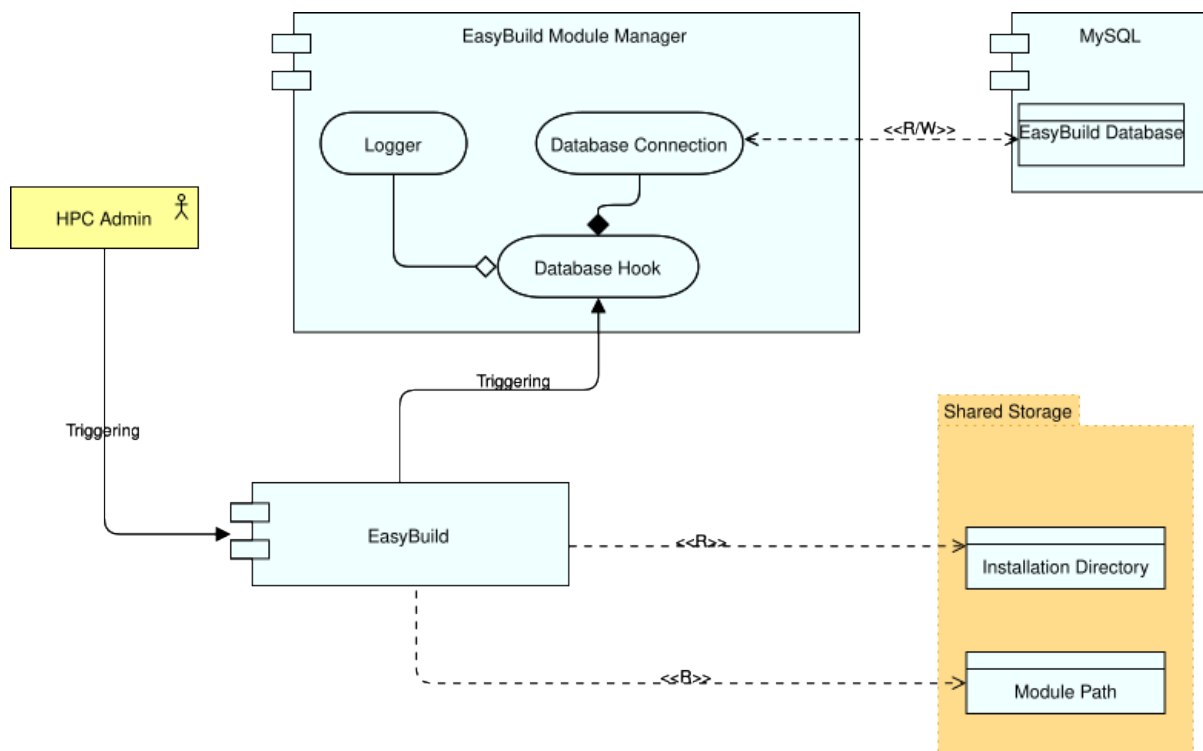


Figure 2: EasyBuild Hook Architecture

Hooks are user-defined functions that are called by the EasyBuild framework at specific times during the installation procedure. They can be leveraged to alter or augment the installation procedure.

The times specified during the installation procedure are:

1. `start_hook`
2. `pre_fetch_hook`
3. `post_fetch_hook`
4. `post_install_hook`
5. `end_hook`

5.1 Start Hook

This hook is invoked upon execution of the command `eb -r MODULE_NAME`. It gathers user-related metadata that remains immutable throughout the installation process, namely, `installed_by` and `requested_by`.

5.2 Pre-Fetch Hook

This hook runs before fetching modules. It collects module data and the dependencies of the module data.

5.3 Post-Fetch Hook

This hook runs after fetching modules. It notifies the user that the module has been successfully fetched.

5.4 Post-Install Hook

This hook runs after installing the module and its dependencies. It collects data of installation `installdir`, adds `installed_by`, `requested_by` to the data of the modules, and notifies the user that the module was successfully installed.

5.5 End Hook

This hook runs after the installation process ends. It inserts the collected data through the previous phases into the database and notifies the user that the module was successfully inserted into the database.

6 TUI (Terminal User Interface)

We have implemented a Terminal User Interface (TUI) utilizing the `curses` library in Python. This choice was made due to the inherent convenience of terminal interfaces, particularly for usage within High-Performance Computing (HPC) Systems, which are commonly accessed via SSH sessions through terminal environments.

When the application is started it opens on the main menu where there is an option to show installed modules.

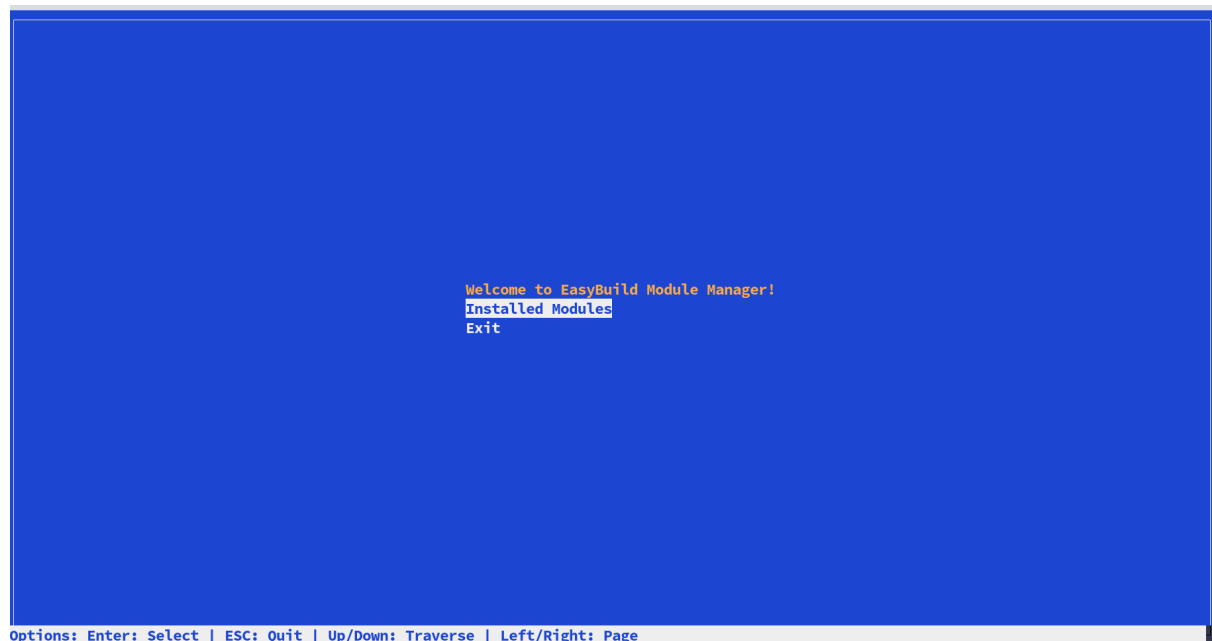


Figure 3: EasyBuild Module Manager Main Menu

After selecting the 'Installed Modules' option from the main menu, the interface will proceed to display a detailed list containing all modules currently installed on the system. This feature

serves to provide users with immediate access to crucial information regarding the system's installed modules, facilitating efficient management and utilization within the HPC environment."



Figure 4: EasyBuild Module Manager Installed Modules List

Upon selecting a specific module from the list and pressing 'Enter', the interface will dynamically generate and display the dependency tree associated with the chosen module. This functionality enables users to visualize the hierarchical relationships between the selected module and its dependencies, offering valuable insights into the interconnectivity of components within the system.

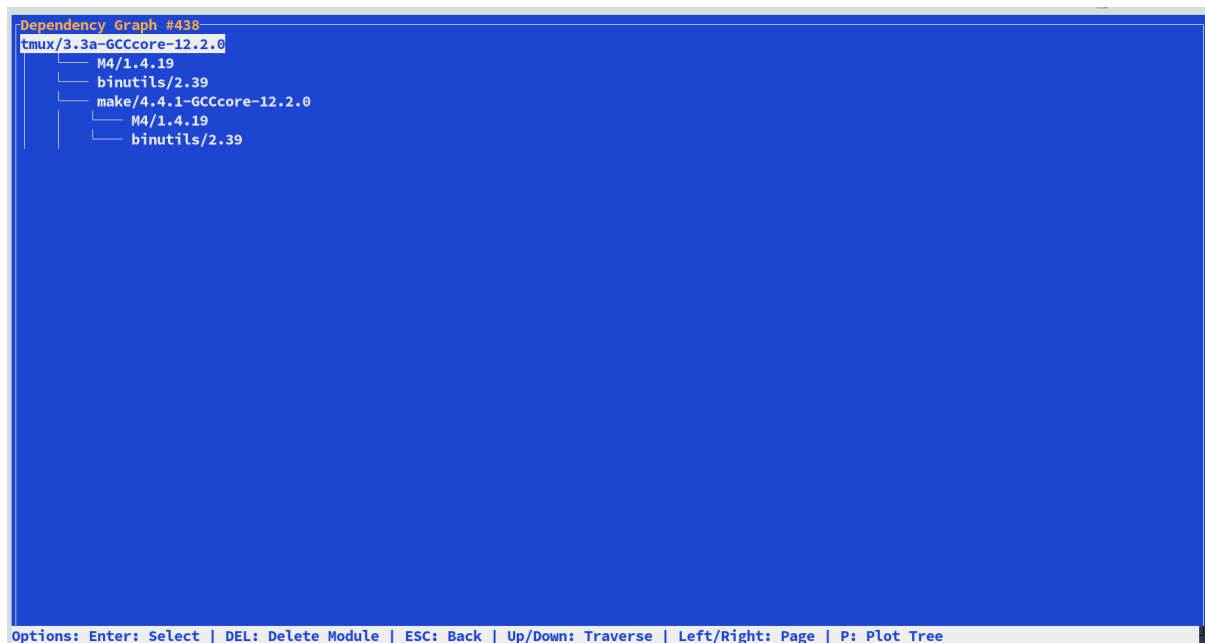


Figure 5: EasyBuild Module Manager Dependency Graph for tmux/3.3a-GCCcore-12.2.0

After navigating the dependency tree and selecting a particular module of interest, users can

access detailed information about the selected module by pressing 'Enter'. This action triggers the display of comprehensive data associated with the chosen module.

```
-Node Data:
Node ID: 3
Name: M4
Full Module Name: M4/1.4.19
Short Module Name: M4/1.4.19
Version: 1.4.19
Version Suffix:
Install Directory:
Toolchain Inherited: 1
System: 1
Hidden: 0
External Module: None
Installed By: eb-admin
Installed At: 2023-12-29 16:15:03
Requested By: HPC
Module Subdirectory:
Load Dependency Modules: None
Dependencies: None
Build Only: 1
Toolchain Dependencies: None
External Module Metadata: {}
Toolchain: {'name': 'system', 'version': 'system'}
Is TC: 1
Build: 1

Options: ESC: Back | Up/Down: Traverse | Left/Right: Page
```

Figure 6: EasyBuild Module Manager List of Data for M4/1.4.19

Additionally, users can delete a module along with its dependencies by pressing the 'DEL' button. Upon initiating this action, the interface prompts users with a confirmation message to ensure intentional deletion. This feature enhances system management efficiency by providing a straightforward method for removing unwanted modules and their associated dependencies.

```
Are you sure you want to delete tmux/3.3a-GCCcore-12.2.0 with its dependencies?
Yes No

Options: Enter: Confirm | Left/Right: Page
```

Figure 7: EasyBuild Module Manager Deletion Confirm Screen to Delete tmux/3.3a-GCCcore-12.2.0

Upon confirming the deletion by selecting "Yes," the interface will present a detailed list highlighting the module and its associated dependencies. Moreover, it will highlight dependencies

that cannot be deleted due to their utilization by other modules. This visual cue informs users about dependencies integral to the functionality of other modules, thereby ensuring informed decision-making during the deletion process.

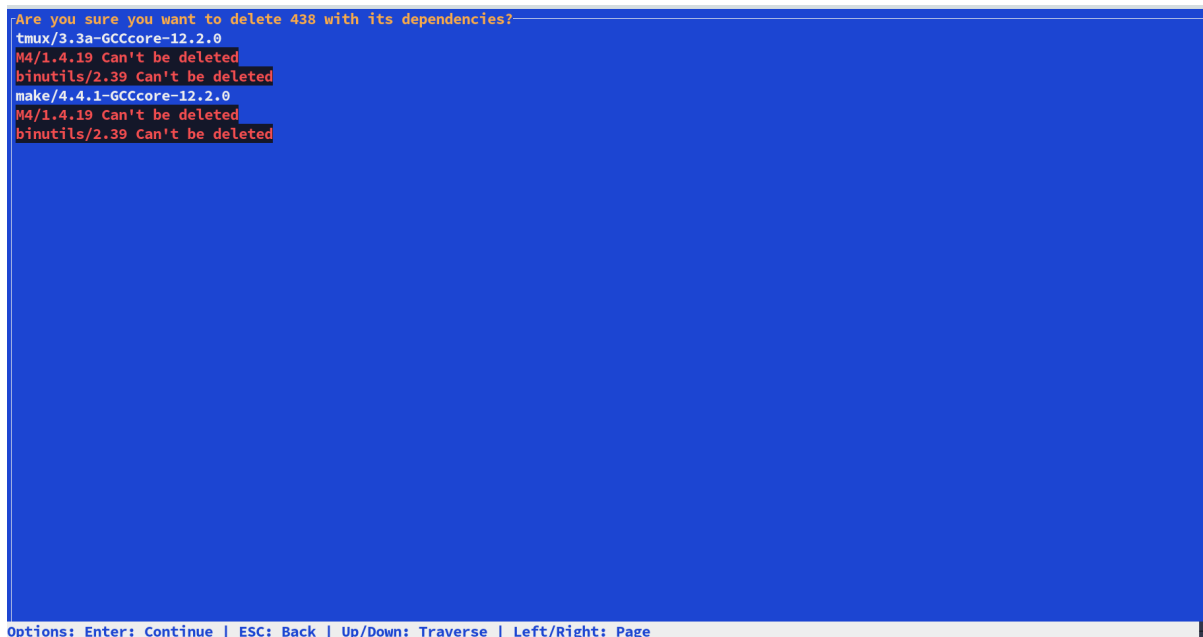


Figure 8: EasyBuild Module Manager List of Module Deletion State Where M4/1.4.19 and binutils/2.39 Won't be Deleted

Following the confirmation of module deletion, the interface initiates the removal process by executing the following actions:

1. Removal from $\$MODULEPATH$: The module is removed from the $\$MODULEPATH$ environment variable, ensuring that it is no longer accessible or loaded by the system.
2. Deletion from Installation Directory: The interface proceeds to remove the module from its installation directory, effectively eliminating all associated files and directories from the system.

These actions guarantee the complete removal of the module from the module path, ensuring a thorough cleanup process and maintaining system integrity.

Also, any dependencies highlighted during the deletion process will remain unaffected. This ensures that essential dependencies, which are utilized by other modules, are preserved and continue to function seamlessly within the system environment. By leaving these dependencies untouched, the interface maintains the stability and integrity of the system while facilitating efficient module management.

After the deletion process is completed successfully, users will be presented with a continuation screen, offering the option to return to the main menu.

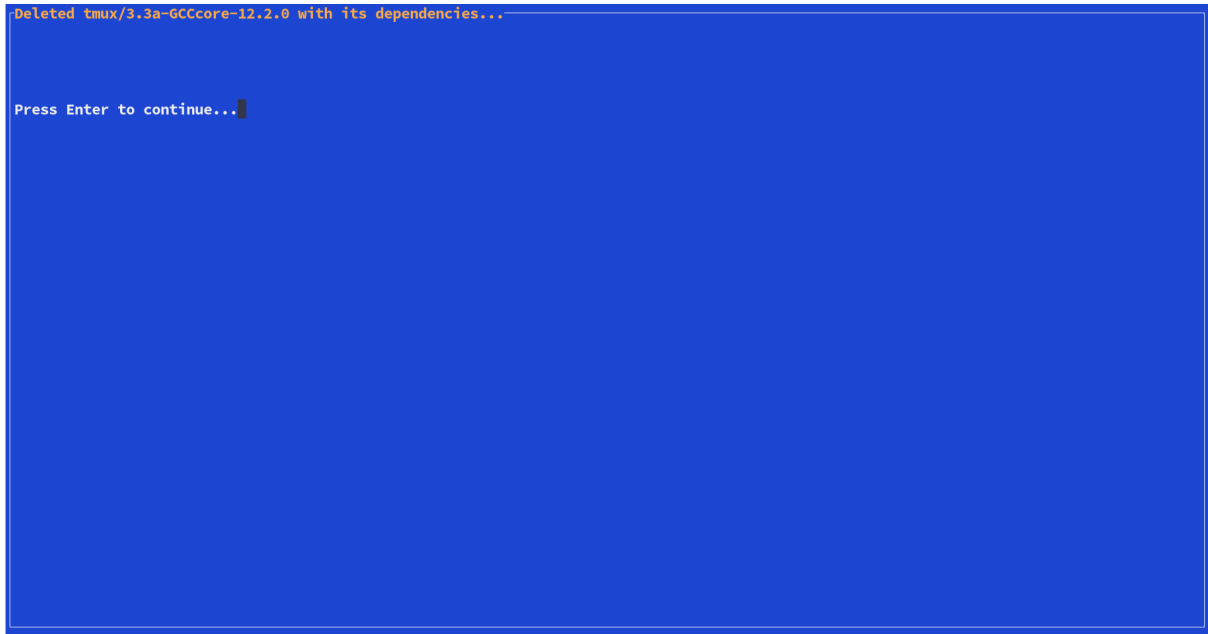


Figure 9: EasyBuild Module Manager Deletion for `tmux/3.3a-GCCcore-12.2.0` Deletion Completed Screen

Following the successful deletion of the module, users can access the installed modules list once again. Upon review, the deleted module will no longer be listed, affirming its successful removal from the system.



Figure 10: EasyBuild Module Manager Installed Modules List After `tmux/3.3a-GCCcore-12.2.0` Deletion Completed

7 Logging

The logger class is a singleton class where the object instance is shared among all the application classes, the logger saves the log in the path specified in the `.env` file '`LOG_PATH`'.

There are 6 distinct levels for the log message:

1. debug
2. info
3. warning
4. error
5. critical
6. custom level

3 types of logging messages are actively used in the application.

7.1 Debug Level

It is mainly used to monitor connection retrieval when a new object is created from the database.

```
1 2024-02-12 10:48:24,842 - my_logger
2 - DEBUG - /easybuildgraphinterface/database.py Returning connection
```

7.2 Info Level

Used to monitor any data retrieval or events that occurred during the show or to delete a module.

```
1 2024-02-12 10:51:56,115 - my_logger
2 - INFO - /easybuildgraphinterface/database.py Retrieved 43 nodes
3
4 2024-02-12 10:53:38,618 - my_logger
5 - INFO - /easybuildgraphinterface/utility.py Graph with root id 42 retrieved
  ↳ successfully.
```

7.3 Error Level

Used to log when an exception is thrown during the runtime.

```
1 2024-02-12 11:08:41,782 - my_logger - ERROR
2 - /easybuildgraphinterface/database.py Error connecting to database:
3 TCP/IP port number should be an integer
4
5 2024-02-12 14:10:39,610 - my_logger - ERROR
6 - /easybuildgraphinterface/module.py An error occurred while deleting the file
7 './modules/libxml2/2.10.3-GCCcore-12.2.0.lua': 'Logger' object has no attribute
  ↳ 'info.'
```

8.1 Architecture

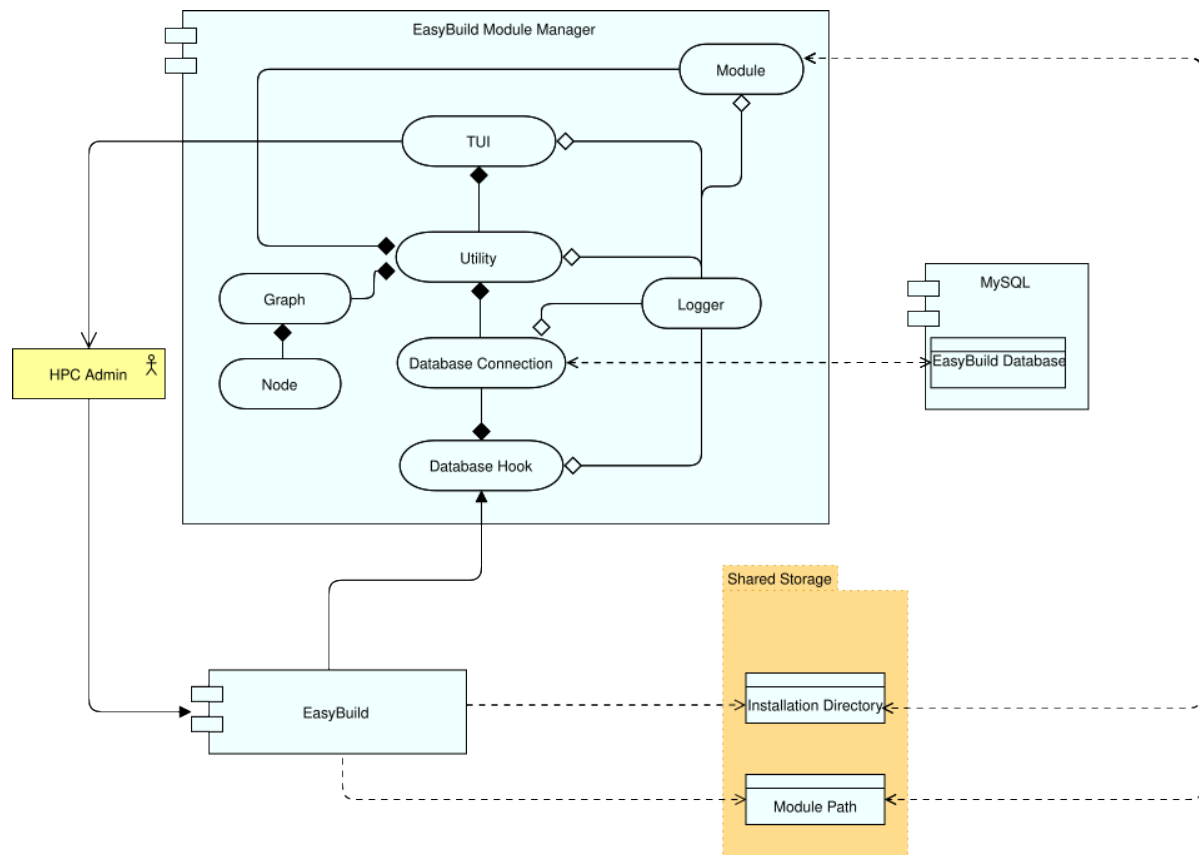


Figure 11: EasyBuild Module Manager Deletion for tmux/3.3a-GCCcore-12.2.0 Deletion Completed Screen

8.2 Module Dependency Tree Diagram

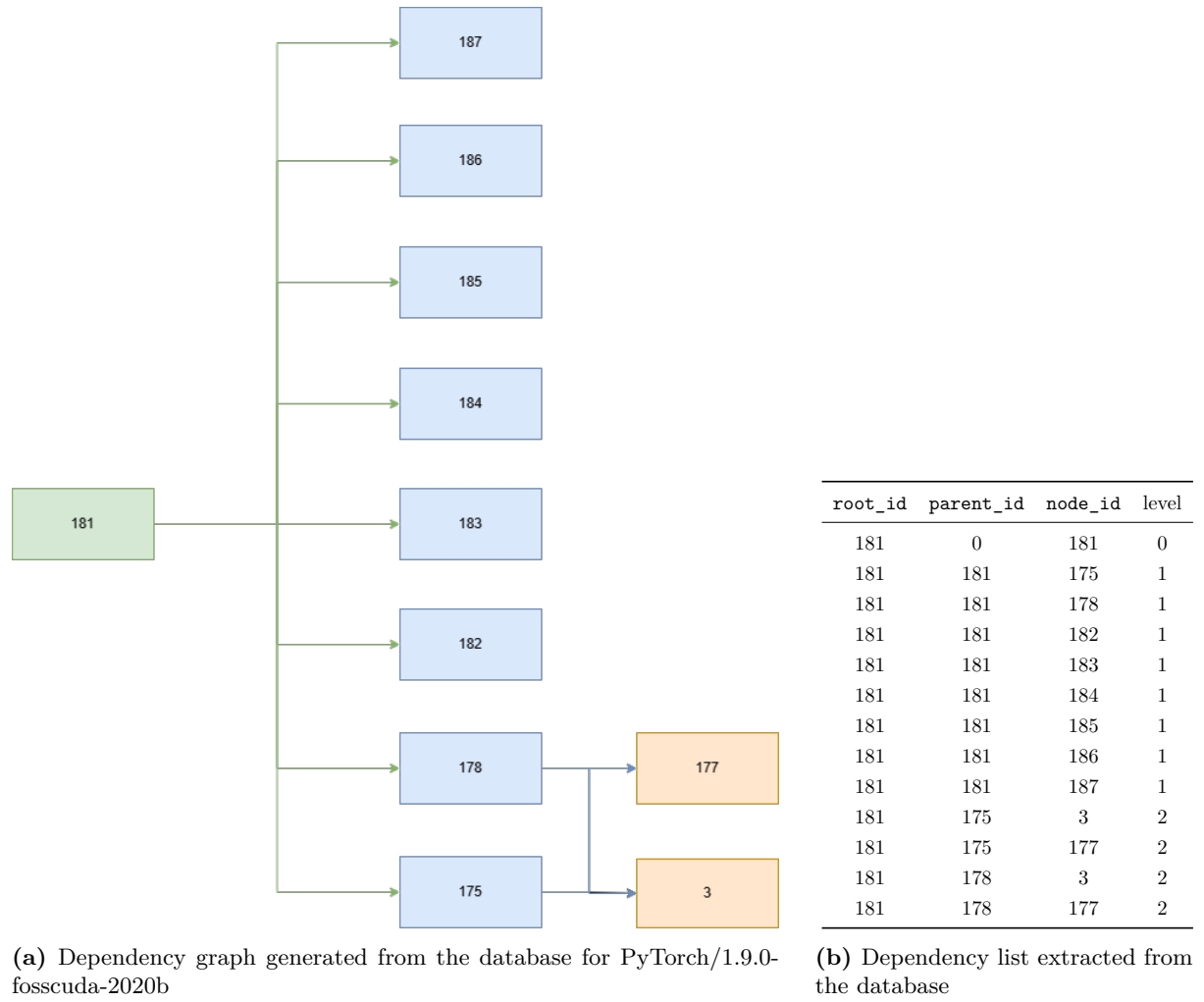


Figure 12: Module dependency tree representation for PyTorch/1.9.0-fossCUDA-2020b

8.3 Class Diagram

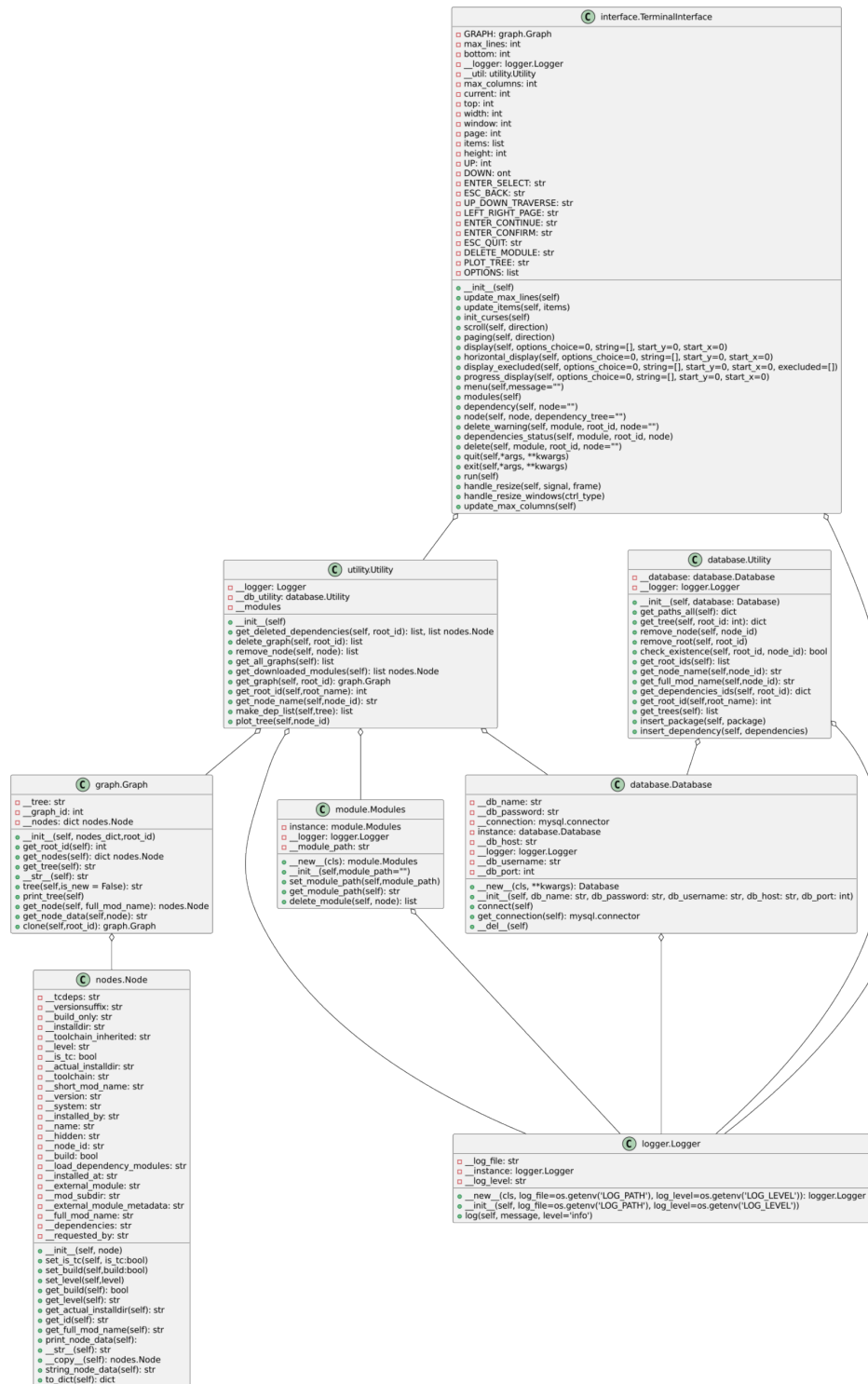


Figure 13: EasyBuild Module Manager Class Diagram

9 Future Works

9.1 Implementing Multi-Tenant Architecture with User Stack[4]

In future iterations of our system, we plan to implement a multi-tenant architecture with a focus on user stacks. This approach involves downloading specific modules for each user or group into their respective home directories, typically located at `$HOME/modules`. Additionally, we aim to integrate a shared database with multiple schemes to support this architecture effectively.

9.1.1 Expected Benefits:

1. Streamlined Management: Users will have an intuitive interface for managing their modules.
2. Efficient Cleanup: Modules associated with discontinued users, deleted accounts, or completed projects can be systematically removed.
3. Enhanced Tracking: Detailed records of module usage by each user or group will be maintained, aiding in system transparency and management.
4. Improved Isolation: User-based environments will be isolated from the global stack, which will exclusively contain modules shared across all users, thereby enhancing system organization and performance.

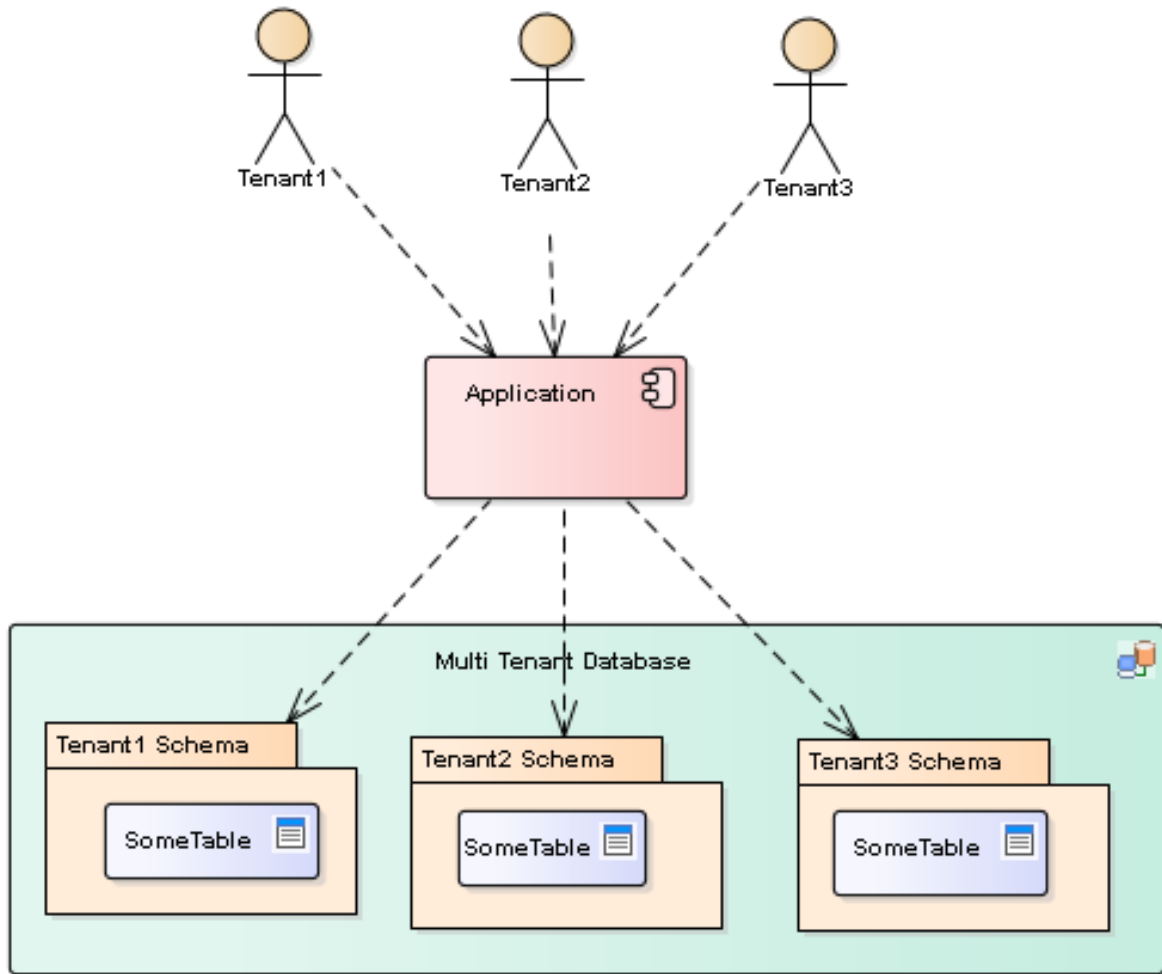


Figure 14: Shared Database, Multiple Schemas Architecture

9.2 Unused Modules

To enhance module tracking efficiency, an alias for the *lmod module load* command, denoted as `MODULE_NAME`, should be established. This alias will ensure that each time a new module is loaded, it will trigger a database counter incrementation in the `load_count` field. Subsequently, a database function will be invoked to update `latest_load_at` utilizing the following SQL query and trigger:

```

1 ALTER TABLE Packages ADD COLUMN latest_load_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
2 ALTER TABLE Packages ADD COLUMN load_count UNSIGNED INT DEFAULT 1;
3
4 CREATE TRIGGER update_timestamp
5 BEFORE UPDATE ON Packages
6 FOR EACH ROW
7 BEGIN
8     IF NEW.load_count <> OLD.load_count THEN
9         SET NEW.latest_load_at = NOW();
10    END IF;
11 END;

```

```
12
13 UPDATE Packages
14 SET load_count = load_count + 1
15 WHERE id IN (
16     SELECT node_id FROM Dependencies WHERE root_id = (
17         SELECT id FROM Packages WHERE full_mod_name LIKE MODULE_NAME
18     )
19 );
```

10 References

References

- [1] Graham Cox. Storing a tree structure in a relational database, 2023. Online article.
- [2] EasyBuild Community. Support uninstalling software in easybuild, 2013. GitHub issue.
- [3] High-Performance Computing Team at Ghent University. Easybuild hooks, 2023. Documentation.
- [4] Harish Somasundar. Database multi-tenancy, 2021. Online article.